tenAsys®

QUICK START

INtime 4.0 Quick Start Guide

September, 2009

TenAsys Corporation
1400 NW Compton Drive, #301
Beaverton, OR 97006 USA
+1 503 748-4720
fax +1 503 748-4730
info@tenasys.com
www.tenasys.com

Copyright © 2005-2009 TenAsys Corporation. 090925
TENASYS, INTIME, and IRMX are registered trademarks of TenAsys Corporation.
Other trademarks and brand names are the property of their respective owners.

This document is protected by US and international copyright laws.

TENASYS, INTIME, and IRMX are registered trademarks of TenAsys Corporation.

† Other companies, products, and brands mentioned herein may be trademarks of other owners.

Information regarding products other than those from TenAsys has been compiled from available manufacturers' material. TenAsys cannot be held responsible for inaccuracies in such material.

TenAsys makes no warranty for the correctness or for the use of this information, and assumes no liability for direct or indirect damages of any kind arising from the information contained herewith, technical interpretation or technical explanations, for typographical or printing errors, or for any subsequent changes in this article.

TenAsys reserves the right to make changes to specifications and product descriptions at any time, without notice, and without incurring any liability. Contact your local TenAsys sales office or distributor to obtain the latest specifications and product descriptions.

Copyright © 2005–2009, TenAsys Corporation, All Rights Reserved

No part of this guide may be copied, duplicated, reprinted, and stored in a retrieval system by any means, mechanical or electronic, without the written permission of the copyright owner.

September, 2009 Edition

INtime 4.0 Quick Start Guide

Contents

Welcome! — Before You Begin **3**
- Notational Conventions 3
- Requirements 3

Installation **4**
- Locating Your License Keys 5

Example #1: The INtime Application Wizard – HelloWorld **6**
- Using the INtime Application Wizard 6
- Introducing the INtime Explorer 10
- Debugging HelloWorld with Visual Studio 12

Example #2: Working Together – Windows and Real-time **14**
- Two Processes – One Application 14
- Creating the Real-Time Process 15
- Creating the Windows Process 18
  - Create the Project and Setup the Environment 18
  - Creating a Graphical User Interface 21
  - Edit the Code 21
- Running the Complete Solution 25

Example #3: The INscope Performance Analyzer **26**
- How Fast is Deterministic? 26
- Fast Does Not Equal Deterministic 27
- A Multi-threaded Example 27
- Trace the Threads With INScope 32

Next Steps **37**

A. Configuring the INtime Kernel **38**
- Node Management 39
- INtime Device Manager 42

B. Debugging With Spider **43**
- Debugging the HelloWorld Example 43

C. Sample Applications **45**

Copyright © 2005-2009, TenAsys Corporation

Figures

Figure 1: Entering License Codes 5
Figure 2: Starting the INtime Application Wizard 6
Figure 3: Creating a New INtime Project 7
Figure 4: Selecting Process Elements 7
Figure 5: Specifying Polling Thread Parameters 8
Figure 6: Wizard Summary Screen 8
Figure 7: Files Generated by the Wizard 9
Figure 8: Starting INtime Explorer 10
Figure 9: Configuring INtime Explorer Options 11
Figure 10: HelloWorld Console Window 11
Figure 11: Terminating the HelloWorld Process 12
Figure 12: Setting a Breakpoint 13
Figure 13: Basic INtime Solution Architecture 14
Figure 14: Selecting the MFC Application Template 18
Figure 15: MFC Application Type Selections 19
Figure 16: Specifying Additional Include Directories 20
Figure 17: NTXData Dialog Box 21
Figure 18: Running the Complete Solution 25
Figure 19: Comparison of Real-time Systems 26
Figure 20: Modifying Thread Parameters 28
Figure 21: Modifying Thread Parameters 28
Figure 22: MultiThread Project Summary 29
Figure 23: MultiThread Application Output 32
Figure 24: INscope Event Trace 33
Figure 25: Zoomed Inscope Trace 35
Figure 26: Intex View of the Multithread App 36
Figure 27: INtime Control Panel Applet 38
Figure 28: Node Management Kernel Tab 39
Figure 29: Node Management System Wide Tab 40
Figure 30: Device Configuration Applet 42
Figure 31: Starting a Spider Debug Session 43
Figure 32: Reaching a Breakpoint in Spider 44

 Copyright © 2005-2009, TenAsys Corporation

Welcome! — Before You Begin

Thank you for your interest in INtime. The INtime RTOS for Windows offers a unique solution for developing embedded real-time applications for the Microsoft® Windows® platform. Your real-time applications can run as part of a system that includes the Windows OS or as a standalone real-time application on dedicated x86 hardware. Best of all, you create real-time applications using the same Visual Studio development tools you now use to create regular Windows applications. This guide will help you become familiar with the INtime development tools. Detailed technical information about the INtime software architecture, kernel, and APIs can be found in the online help files and user manuals.

Notational Conventions

This guide uses the following conventions:

- All numbers are decimal unless otherwise stated.
- Bit 0 is the low-order bit. If a bit is set to 1, the associated description is true unless stated otherwise.
- `Data structures and syntax strings appear in this font.`

Notes indicate important information about the product.

Tips indicate alternate techniques or procedures that you can use to save time or better understand the product.

Requirements

The tutorial applications in this guide will be built and executed directly on your development machine. Your development machine needs to meet the following minimum requirements:

- Pentium class (or better) processor
- 16MB of free RAM for INtime and your real-time applications
- 75MB hard disk space for tools, examples, and documentation
- Windows 7, Windows Vista, Windows Server 2008, Windows Server 2003, or Windows XP with Service Pack 3
- Visual Studio (Visual Studio 2008, 2005).

INtime applications can run under Windows 7, Windows Vista, Windows XP, Windows XP Embedded, Windows 2003 Server and Windows 2008 Server. The examples in this guide focus on the Windows Vista and Windows XP environments. Check the installer readme file for any amendments to these requirements.

Copyright © 2005-2009, TenAsys Corporation

Installation

The procedure below describes installation of the INtime development tools and kernel on your development machine. Make sure that your development machine meets the requirements listed in the *Requirements* section above.

> *Note:* Install Visual Studio *before* installing INtime and insure that you are logged on with Administrator privileges. If you install Visual Studio after installing INtime, use the INtime Configuration Manager to add the INtime development tools to Visual Studio.

Insert the INtime CD-ROM. If AutoRun is enabled a welcome dialog appears. If AutoRun has been disabled you can start the installation procedure by double-clicking the file named *launch.exe* in the root directory of the INtime CD-ROM. The installation procedure is similar to that of most standard Windows applications. You will be asked to accept the INtime software license agreement and to supply one or more registration keys to complete the installation procedure.

A default destination folder of *%ProgramFiles%\INtime* (typically *C:\Program Files\INtime*) is suggested by the installation program for the INtime development tools and sample files. Make note of this directory so you can locate it again if you wish to inspect header files and other INtime system files. Sample projects will be installed in the user directory of the user who installed INtime. Configuration files will be installed in the *%AllUsersProfile%\Application Data\TenAsys\INtime* directory. On Windows versions after Windows XP the path is *%AllUsersProfile%\TenAsys\INtime*

Two basic install options are provided: target and development. The development installation (referred to as *Development & Local Target* on the *Select Installation Type* dialog box) is a complete installation which includes tools, documentation, examples, and the INtime kernel; everything you need to create and run INtime applications on your development workstation. The target installation (referred to as *Local Target only*) installs only those files needed to run INtime applications on a finished real-time Windows system. A target installation cannot be used for editing and compiling INtime source code. In a development environment, a target installation is typically used to build a system meant to emulate a deployed system.

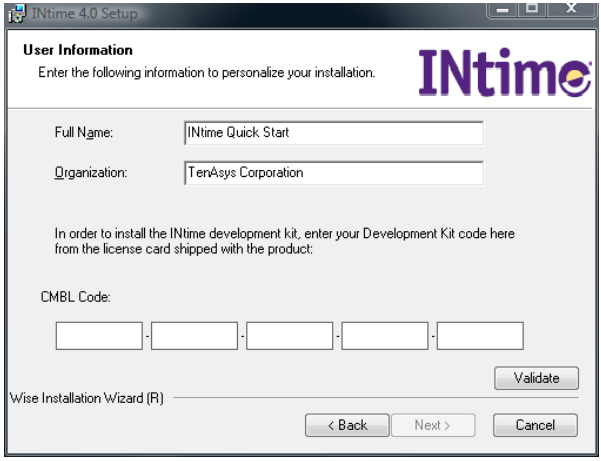
> *Note:* To run the examples in this guide you must select the *Development & Local Target* installation option.

Copyright © 2005-2009, TenAsys Corporation

Locating Your License Keys

Depending on the product you purchased, you will be prompted for a combined development license code (CMBL) or a target license code (TRGL). These license codes can be found on the license card that came with your software.

Figure 1: Entering License Codes


INtime 4.0 Setup
User Information
Enter the following information to personalize your installation.
INtime
Full Name:
INtime Quick Start
Organization:
TenAsys Corporation
In order to install the INtime development kit, enter your Development Kit code here from the license card shipped with the product:
CMBL Code:
Validate
Wise Installation Wizard (R)
< Back
Next >
Cancel

If you are installing a network licensed product you will be prompted to enter the address or name of the license server. Contact your company's IT personnel for the license server information you will need to complete this licensing step.

After installation has completed you will be asked to reboot the system. If you have been provided a hardware key, insert it before rebooting. Once the system has rebooted you have the option of configuring the INtime runtime environment and services. For the purposes of this document the default configuration will suffice.

If you cannot locate your license card please contact TenAsys directly. Contact information is available at www.tenasys.com.

Copyright © 2005-2009, TenAsys Corporation

Example #1: The INtime Application Wizard – HelloWorld

This exercise introduces the INtime Application Wizard used to create a simple real-time process. The INtime Application Wizard assists by adding template code for elements like semaphores, threads, shared memory allocation, interrupt handling, and client threads to a new project. The INtime Application Wizard will be used to create the foundation for the *HelloWorld* example project.

In the *HelloWorld* project you will create a thread that executes in an infinite loop. The thread will sleep approximately 1000 milliseconds and print the phrase "HelloWorld" ten times in an INtime console window, per each loop iteration.

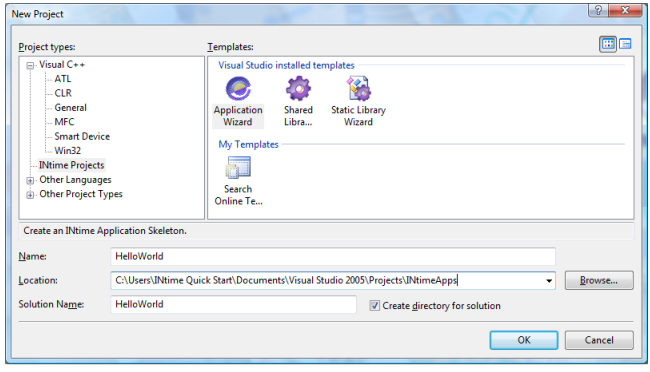
> *Note:* For the sake of brevity, only screenshots of significant value are shown within the tutorials of this guide.

Using the INtime Application Wizard

Create a directory on your development machine called *INtimeApps* to store the examples from this Guide.

1) Start Visual Studio.
2) Select File | New | Project to create a new Visual Studio project.
3) Under INtime Projects, select *INtime Application Wizard*.
4) Enter a project name of *HelloWorld* and set the location (path) to the *INtimeApps* directory you created above.

Figure 2: Starting the INtime Application Wizard



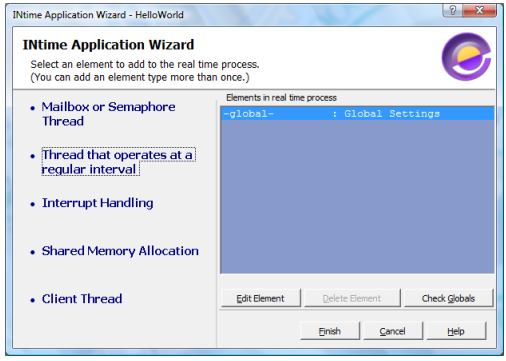
Copyright © 2005-2009, TenAsys Corporation

5) Select *A full-featured application* in the wizard dialog box and leave the C++ box unchecked. We are not using the INtime wizard's *Hello World* application because the features of our sample project are going to be more interesting. Click *OK* to continue.

Figure 3: Creating a New INtime Project

6) The next screen is used to add elements to your real-time process, such as mailboxes, semaphores, and threads. These elements can be created manually; but, using the INtime Wizard saves time and minimizes errors. Select *Thread which operates at a regular interval* from the list of real-time process elements.

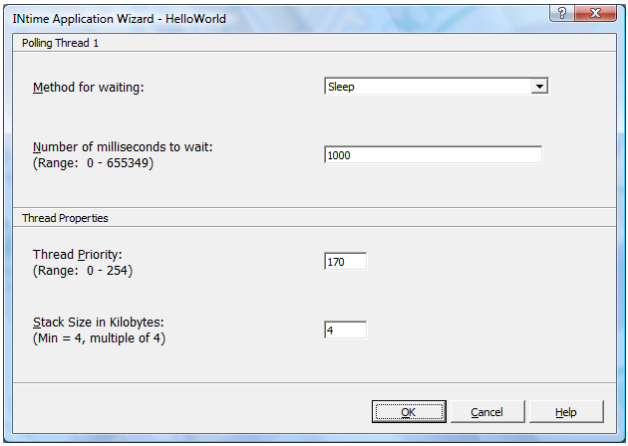
Figure 4: Selecting Process Elements



Copyright © 2005-2009, TenAsys Corporation

7) Keep the default settings for the polling thread, so the thread will wake up every 1000 milliseconds. Click *OK* to return to the *Add Elements* screen.

Figure 5: Specifying Polling Thread Parameters

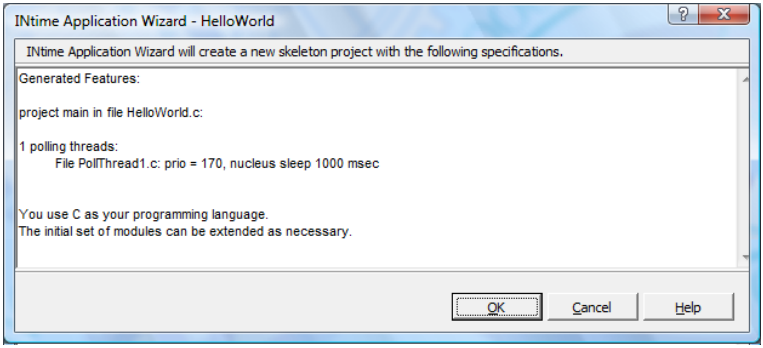


8) Highlight *-global-* in the elements list on the right of the *Add Elements* dialog and click the *Edit element* button. A dialog box is displayed that can be used to modify real-time process parameters. The default parameters are fine for this example.

9) Click *OK* and then the *Finish* button.

10) The final wizard summary screen appears. Press *OK* on this screen to generate the project files.

Figure 6: Wizard Summary Screen

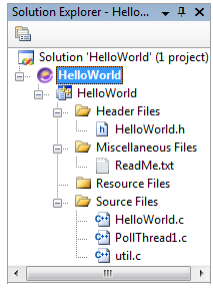

INtime Application Wizard - HelloWorld
INtime Application Wizard will create a new skeleton project with the following specifications.
Generated Features:
project main in file HelloWorld.c:
1 polling threads:
File PollThread1.c: prio = 170, nucleus sleep 1000 msec
You use C as your programming language.
The initial set of modules can be extended as necessary.
OK
Cancel
Help

Copyright © 2005-2009, TenAsys Corporation

Visible in the Visual Studio solutions explorer are three .C files that were generated be the INtime Application Wizard:

- *HelloWorld.c*: the `main()` function which contains initialization and cleanup code. The file name is derived from the project name.
- *Pollthread1.c*: the polling thread code generated by the *add real-time elements* section of the wizard.
- *Util.c*: contains general-purpose utility routines.

Figure 7: Files Generated by the Wizard



1) Open *Pollthread1.c*.
2) Add an integer named 'x' at the start of the polling thread.
3) Add a `for` loop and `printf()` statements after the TODO comment. The resulting code should look like the following (additions are shown in bold):

```
void            PollThread1(void)
{
    int x;

#ifdef _DEBUG
    printf("PollThread1 started\n");

#endif
    while (1)
    {
        RtSleep(1000);

#ifdef _DEBUG
        printf("PollThread1 waking up\n");
#endif

        // TODO: do what has to be done every 1000 milliseconds
        // Print HelloWorld 10 times
        for(x=0; x<10; x++)
            printf("HelloWorld!\n");
    }
}
```

Make sure the build type is set to *Debug* (Build | Configuration Manager…) and build the solution (Build | Build Solution). The *HelloWorld* program should compile and link without errors.

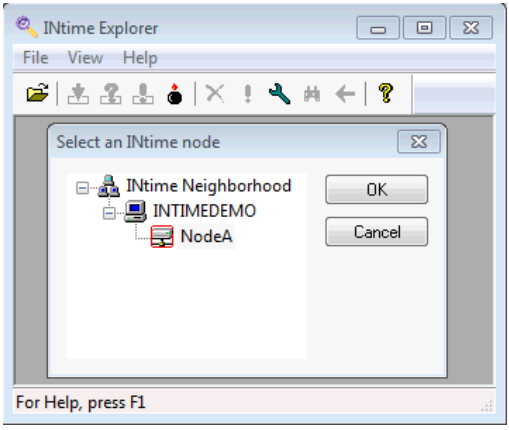
Copyright © 2005-2009, TenAsys Corporation

Introducing the INtime Explorer

Double-clicking an RTA file, when the INtime kernel is running, loads and runs the real-time application (RTA); we will use the INtime Explorer (aka INtex) to load and run the *HelloWorld* example that was just built. INtex displays the real-time objects present on an INtime node (an INtime real-time kernel).

1) Load the INtime kernel, if it is not already running; from the INtime Status Monitor in the Windows system tray, select NodeA | Start NodeA. Note that *NodeA* is the default name of the INtime kernel, you can create other nodes with different names.
2) Start INtime Explorer using its shortcut in the INtime program group.
3) Select the *NodeA* node from the dialog box, and press *OK*.

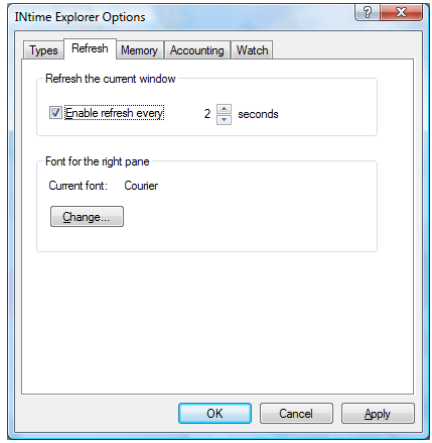
Figure 8: Starting INtime Explorer



Copyright © 2005-2009, TenAsys Corporation

4) When debugging a *local* INtime node, it is useful to turn on the INtex automatic refresh. Go to View | Options… on the INtex menu and select the Refresh tab. Check the *Enable refresh every* box and set the interval for two seconds.

Figure 9: Configuring INtime Explorer Options

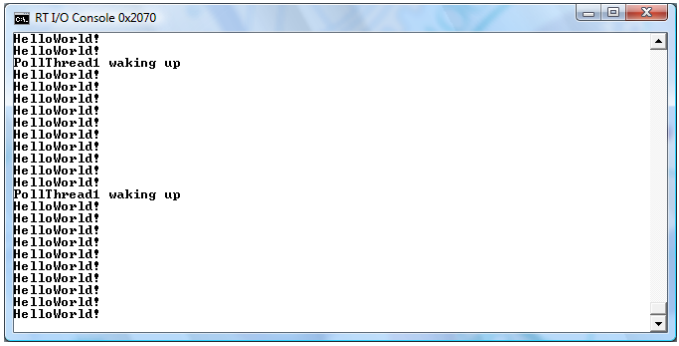


5) To load and run *HelloWorld* click the second button on the INtex toolbar or use the File | Load RT app menu.

6) Navigate to the debug folder in your *HelloWorld* project directory and select the real-time executable file *HelloWorld.rta*. Pushing the *Open* button loads and starts the real-time process on the INtime kernel.

A console window should now be visible and the message *HelloWorld!* should appear ten times each second inside the console window.

Figure 10: HelloWorld Console Window

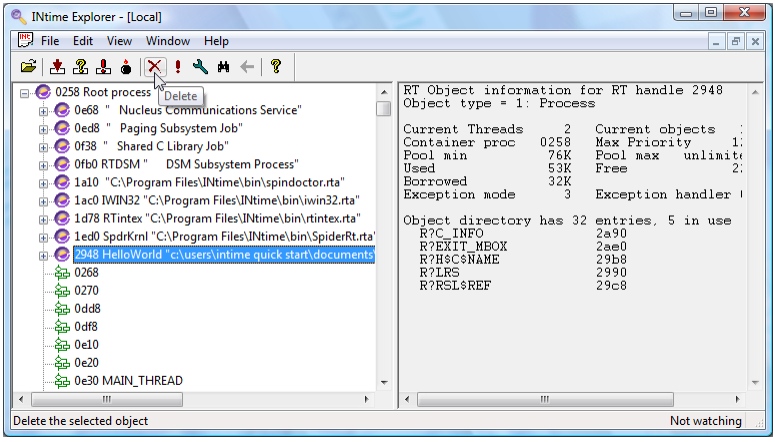


Copyright © 2005-2009, TenAsys Corporation

Terminate the *HelloWorld* process by doing the following:

1) Go to the INtime Explorer main window.
2) Find the *HelloWorld* real-time process in the left window of INtime Explorer (each INtime icon represents one real-time process).
3) Click the line to select the HelloWorld process.
4) Click the red 'X' button in the toolbar to delete the process

Figure 11: Terminating the HelloWorld Process



Answering *Yes* to the deletion warning pop-up terminates the real-time process. The *HelloWorld* process icon disappears from the INtime Explorer process list. Notice that the *HelloWorld* console window remains on your desktop, but the title bar of the console window displays *Finished*.

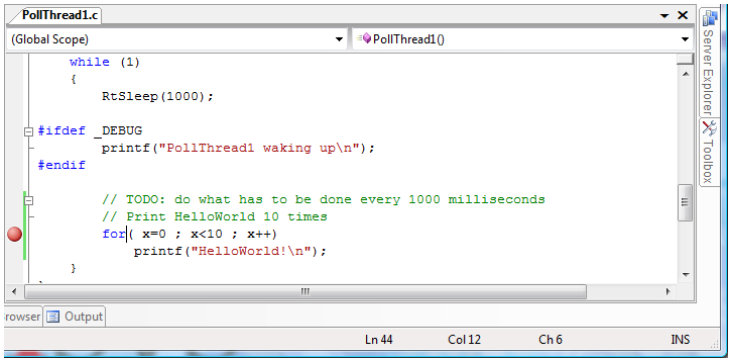
Debugging HelloWorld with Visual Studio

INtime includes the ability to debug real-time processes directly from within Visual Studio 2005 and Visual Studio 2008. Using the *HelloWorld* project we just created, you can step through the code and perform basic debugging tasks.

1) If the INtime kernel is not already running, start it using INtime Status Monitor | NodeA | Start NodeA.
2) Start the Visual Studio development environment.
3) Open the *HelloWorld* project.
4) Open *Pollthread.c* within the *HelloWorld* solution.
5) Set a breakpoint on the `for` loop, either by double-clicking the vertical bar to the left of the source window or by placing the cursor on the line and pressing the *F9* key.

 Copyright © 2005-2009, TenAsys Corporation

Figure 12: Setting a Breakpoint



6) Start the debugger by pressing the *F5* key, or by clicking the start button on the Visual Studio tool bar. This will launch the *HelloWorld.RTA* process.
Note: if you are not running on the default configuration, you may need to select the target INtime node in the INtime project settings.

7) The *HelloWorld* process will run to the breakpoint. Following the break, you can step through the code and watch variables change (e.g., 'x') as you step through the loop. Debugging an INtime real-time thread in this way is virtually identical to debugging a Windows thread.

Copyright © 2005-2009, TenAsys Corporation

Example #2: Working Together – Windows and Real-time

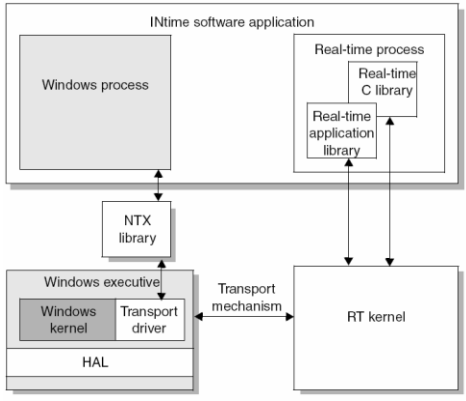
The typical INtime solution consists of two executables: a standard Windows process that provides access to the Windows user interface, database functions, and other Windows-specific functions and a real-time INtime process containing time-critical threads. Communication between the two parts is managed by the INtime NTX library.

In this example we will use INtime *data mailbox* objects to demonstrate how a simple Windows MFC dialog process can exchange data with real-time threads running on the INtime kernel.

Two Processes – One Application

Three data mailboxes, *MY_MBOX_1*, *MY_MBOX_2*, and *MY_MBOX_3* will be used to send data between two processes: *NTXData.exe* (a Windows process) and *RTData.rta* (a real-time process) and to signal between two real-time threads. Together these two processes comprise a single INtime software application.

Figure 13: Basic INtime Solution Architecture



This data mailbox example is only one possible solution for sharing data between an INtime real-time application and a Windows application; other solutions might incorporate shared memory or exchanging semaphores between Windows processes and real-time processes.

> *Tip:* To learn more locate the topic "Understanding the INtime Software Architecture" in the INtime User's Guide.

Copyright © 2005-2009, TenAsys Corporation

Creating the Real-Time Process

First we will create *RTData.rta*, the real-time process that will be launched by *NTXData.exe*. The real-time process will setup the mailboxes and wait for the Windows process to send a data message through the first mailbox. After a message is received, the real-time process will send data back to the Windows process using the second mailbox. The third mailbox will be used for internal communication between real-time threads within the real-time process.

1) Open Visual Studio and create a real-time project called *RTData* and place it in the *INtimeApps* directory you created for the *HelloWorld* example.

> *Note:* It is important that this project be named "*RTData*," exactly as shown above. The name you specify here is used as a real-time process object identifier and will be referenced in later code. INtime object names are case-sensitive.

2) Choose *A full-featured application* from the INtime Application Wizard and click *OK* (leave *C++* unchecked for this example).
3) Add a *Data mailbox* element by selecting *Mailbox or Semaphore Thread* from the list of available elements. (The *Type of object this thread waits at* should be set to *Data mailbox* in the options dialog.)
4) Type *MY_MBOX_1* for the *Catalog the object with this name* field.
5) Click *OK* to return to the elements setup dialog.
6) Repeat the preceding three steps to add a second data mailbox, but this time name it *MY_MBOX_2*.
7) From the elements setup dialog add a *Client Thread* (last element in the list).
8) Check the *Send to data mailbox* item (upper left) and leave all other items unchecked, then click *OK* to return to the elements setup dialog.
9) Click *Finish* followed by *OK*; the wizard will automatically generate real-time code templates.

The code generated by the above steps is only a starting point. Modifications are required to turn this project into a running program. Data mailbox *MY_MBOX_1* will receive messages from the Windows process, and data mailbox *MY_MBOX_2* will send messages to the Windows process.

The client thread in *ClientThread1.c* will send messages to the Windows process via *MY_MBOX_2*. The code in *MY_MBOX_2.c* will

Copyright © 2005-2009, TenAsys Corporation

only be used to create that data mailbox. In addition, we will manually add a third data mailbox for inter-thread communication.

> *Note:* This file and function structure is not necessarily the most efficient or elegant solution; it is being used to quickly demonstrate the INtime architecture and the use of INtime wizards to generate template code.

Make the modifications shown below in bold to *RTData.c*. This file contains the real-time process' `main()` function. These modifications add a third data mailbox to coordinate receiving data from *MY_MBOX_1* and sending data via *MY_MBOX_2*. The last lines we add take control of the region object and release that control after thread initialization is complete.

> *Tip:* Open the electronic (PDF) version of this Guide and use the Adobe Acrobat "Text Tool" to copy and paste these code fragments directly from the documentation into your Visual Studio project.

```
// global variables
    RTHANDLE    hRootProcess;
    DWORD       dwKtickInUsecs;
    RTHANDLE    hMBOX_Signal;

                …intervening lines removed for brevity…

    // create mailbox and semaphore threads
    hMBOX_Signal = CreateRtMailbox(DATA_MAILBOX | FIFO_QUEUING);
    if (hMBOX_Signal == BAD_RTHANDLE)
        Fail("Cannot create signaling data mailbox");
```

Do not forget to include a global declaration for the region object, `hMBOX_Signal`, at the end of *RTData.h*.

```
extern RTHANDLE     hRootProcess;
extern DWORD        dwKtickInUsecs;
extern RTHANDLE     hMBOX_Signal;
```

Open *MY_MBOX_1.c*. The Wizard has generated code to create, initialize, and retrieve data from the mailbox. We are adding code to print received data to a console window and signal to *ClientThread1* that a reply message can be sent.

```
        // TODO: operate on byMmessage (and dwActual)
        //Print the message received from the mail box
        printf("This is the message: %s\n", byMessage);

        //Indicate that the message was received
        SendRtData(hMBOX_Signal, "go", 3);
    }
}
```

Next, open *MY_MBOX_2.c* and remove the lines in the `while` loop that wait for data to be received from the data mailbox; in the code

 Copyright © 2005-2009, TenAsys Corporation

fragment below they have been commented out. In this example we will only use this thread to initialize the data mailbox.

The Windows process will receive from this data mailbox and the real-time process will send through this data mailbox. Add a line at the end of the `while` loop to suspend the thread.

```
//    wActual = ReceiveRtData(hMY_MBOX_2, byMessage, WAIT_FOREVER);
//    if (wActual == 0)
//      Fail("Receive from data mailbox MY_MBOX_2 failed");

    // TODO: operate on byMmessage (and dwActual)
    SuspendRtThread(GetRtThreadHandles(THIS_THREAD));
```

Finally, open *ClientThread1.c* and add the `retMessage[]` array to be used for building return messages. Remove the lines being used to lookup the process handle, since the data mailbox we will reference in this thread was created in this process. And, modify the parameters accordingly in the line that gets the handle to the *MY_MBOX_2* data mailbox.

```
void ClientThread1(void)
{
    RTHANDLE    hProcess;
    RTHANDLE    hDmbx;
    char        retMessage[128];
    int         x = 0;
    int         y;
    // TODO: adjust process and mailbox name
    // TODO: remove the next lines if the data mailbox
    // was created in this process
//    hProcess = LookupRtHandle(hRootProcess, "DMBX_OWNER",
WAIT_FOREVER);
//    if (hProcess == BAD_RTHANDLE)
//        Fail("Cannot find data mailbox process");

    // TODO: replace hProcess by NULL_RTHANDLE
    // if the data mailbox was created in this process
//    hDmbx = LookupRtHandle(hProcess, "DMBX_NAME", WAIT_FOREVER);
    hDmbx = LookupRtHandle(NULL_RTHANDLE, "MY_MBOX_2",
WAIT_FOREVER);
    if (hDmbx == BAD_RTHANDLE)
        Fail("Cannot find data mailbox");
```

Finally, add code in the `while` loop to wait for the signal indicating that we should send a message to the Windows process. We will assemble the message sent by including an incremented count value so each response message is unique.

```
    while (1)
    {
        // TODO: put client code that must be repeated here
        // the RtSleep call is just an example
//        RtSleep(1000);
        ReceiveRtData(hMBOX_Signal, retMessage, WAIT_FOREVER);

//        if (!SendRtData(hDmbx, "test", 5))
        y = sprintf(retMessage, "%s %i", "Msg rcvd: ", x++);
        if (!SendRtData(hDmbx, retMessage, ++y))
            Fail("Cannot send to data mailbox");
    }
```

We are ready to build the application. Choose Build | Build Solution from the Visual Studio menu to compile and link. Check the *Debug* folder in your *RTData* project directory and you should find an *RTData.rta* file, among others. This is the real-time process' executable (equivalent to a Windows EXE file). Ignore any warnings regarding unreferenced local variables, these are vestiges of code we deleted in the edits above.

Creating the Windows Process

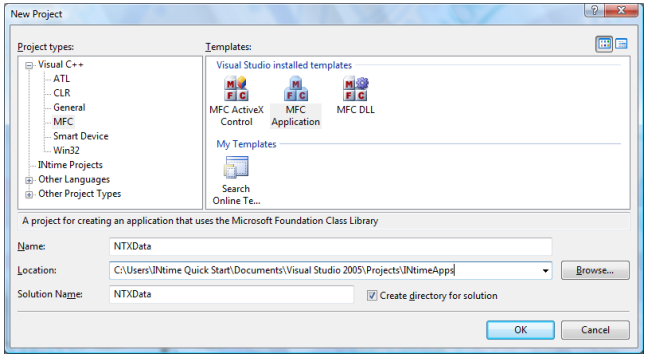
When we create the Windows part of our application, we need to setup the project environment so it includes NTX support. Creating the Windows application will take a few steps.

Create the Project and Setup the Environment

> *Note:* These instructions are specific to Visual Studio 2005, but should also work for 2008.

1) Start Visual Studio.
2) From the menu, select File | New | Project.
3) Within *Visual C++ Projects* open the *MFC* folder and select the *MFC Application* template.
4) Specify *NTXData* as the project name and save the project to your *INtimeApps* folder.

Figure 14: Selecting the MFC Application Template

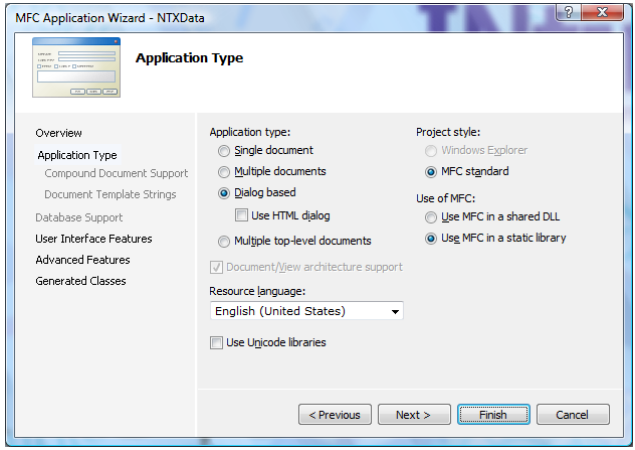


Copyright © 2005-2009, TenAsys Corporation

5) In the *MFC Application Wizard* dialog box select *Application Type* (on the left side of the dialog box).

6) Within the *Application Type* options select *Dialog based* under *Application type* and *Use MFC in a static library* under *Use of MFC*. Ensure that "Use Unicode libraries" is *not* selected (If you wish to use Unicode libraries you should not use the _T() function when passing text into functions requiring LPSTR)

Figure 15: MFC Application Type Selections



7) Click *Finish* and the Wizard will generate template code.

Before we proceed with the MFC code we need to modify the project properties to include the NTX library and header files.

> *Note:* These instructions are specific to Visual Studio 2005, but should also work for 2008.

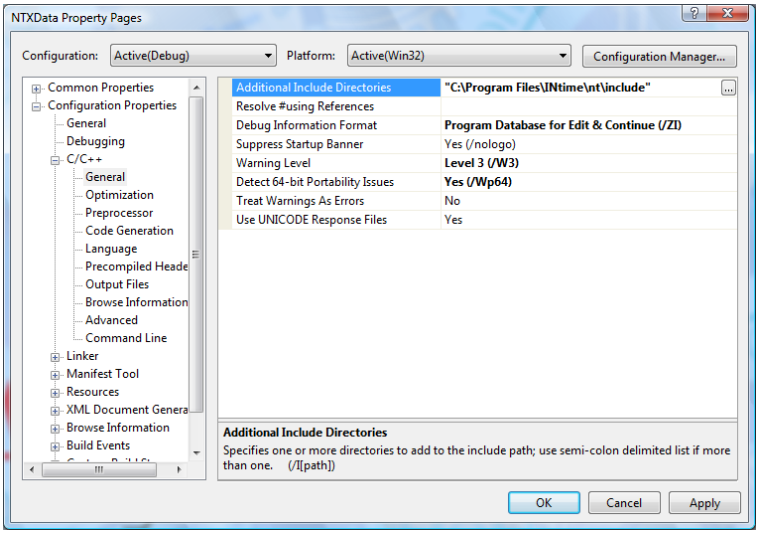
1) From the Visual Studio menu select Project | NTXData Properties…

2) On the property pages dialog choose *All Configurations* in the *Configuration* pull down.

3) Select *General* from the *C/C++* category of the property list and type *C:\Program Files\INtime\nt\include* in the *Additional Include Directories* field (substitute the appropriate directory name if you installed the INtime program files into a different location).

Copyright © 2005-2009, TenAsys Corporation

Figure 16: Specifying Additional Include Directories



4) Similarly, in the *General* section of the *Linker* category type *C:\Program Files\INtime\nt\lib* in the *Additional Library Directories* field (substitute the appropriate directory name if you installed the INtime program files in a different location).

5) Under the *Input* section of the *Linker* category type *ntx.lib* in the *Additional Dependencies* field.

6) Click *OK* to save changes and close the property pages dialog.

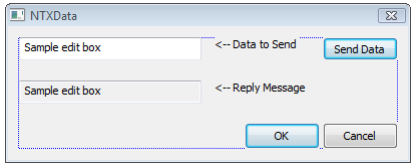
Copyright © 2005-2009, TenAsys Corporation

Creating a Graphical User Interface

Now we will create the GUI for the Windows process.

1) Remove the *TODO: Place dialog controls here* text object.
2) From the *Dialog Editor* in the *Toolbox* bar add two *Edit Control* objects, two *Static Text* objects, and one *Button* object. The figure below shows a layout for the controls in NTXData's dialog box.

Figure 17: NTXData Dialog Box



3) Modify the properties of each control as follows:

IDC_Button1	
ID	IDC_txDATA
Caption	Send Data
Default Button	True

IDC_Edit1	
ID	IDC_DATA

IDC_Edit2	
ID	IDC_rxDATA
Read Only	True

IDC_Static2	
ID	IDC_STDATA
Caption	Data to Send

IDC_Static3	
ID	IDC_STRM
Caption	Reply Message

Leave the *OK* and *Cancel* buttons as part of the dialog box. You will be able to use them to close the NTXData application.

Edit the Code

We will add code to start the *RTData.rta* process when *NTXData.exe* starts, using the INtime NTX API.

1) Declarations for the real-time handles must be marked as protected. Open the *NTXDataDlg.h* header file.
2) Add a `#include "ntx.h"` line at the top of the file.
3) In the protected section of the class definition add declarations for the handles needed to locate the *RTData* process and access the data mailboxes.

Copyright © 2005-2009, TenAsys Corporation

```
// NTXDataDlg.h : header file
//

#include "ntx.h"

#pragma once

…intervening lines removed for brevity…

// Implementation
protected:
    HICON m_hIcon;

    //The handles to the root processes, RTData, and mail box
    NTXHANDLE   m_RootProcess;
    NTXHANDLE   m_TestNTXProcess;
    NTXHANDLE   m_RtMailbox_1, m_RtMailbox_2;

    // Generated message map functions
    virtual BOOL OnInitDialog();
    afx_msg void OnSysCommand(UINT nID, LPARAM lParam);
    afx_msg void OnPaint();
    afx_msg HCURSOR OnQueryDragIcon();
    DECLARE_MESSAGE_MAP()};
```

4) Open *NTXDataDlg.cpp*.

When *NTXData.exe* starts it will load the *RTData.rta* application.

5) In `CNTXDataDlg::OnInitDialog` add the following initialization code to load and start the *RTAData.rta* application.

```
BOOL CNTXDataDlg::OnInitDialog()
{
    NTXHANDLE       hNtx, hRemoteApp;
    CString         tmp;

…intervening lines removed for brevity…

    // TODO: Add extra initialization here

    //Launch RTA Application
    //First set the location of the node
    //Typically you would want to use a browser
    //to select from the available nodes
    //For this example we will hard code the node to Local
    hNtx = ntxGetLocationByName("Local");

    //check to see that the node is there
    if( ntxGetRtStatus( hNtx ) != E_OK ) {
        MessageBoxEx(NULL, _T( "RT Machine not present" ),
            _T( "NTXData"),
            MB_ICONERROR | MB_OKCANCEL, LANG_ENGLISH );
        exit(0);
    }

    //Now launch the RTData.rta application
    hRemoteApp = ntxCreateRtProcess(
        hNtx,_T("C:\\INtimeApps\\RTData\\Debug\\RTData.rta"),
        NULL, NULL, NTX_PROC_SHOW_PROGRESS);
    if (hRemoteApp == NTX_BAD_NTXHANDLE) {
        tmp = "Cannot load file";
        MessageBox(tmp);
```

 Copyright © 2005-2009, TenAsys Corporation

```
        EndWaitCursor();
        exit(0);
    }
    return TRUE;    // return TRUE  unless you set the focus to a
control
}
```

> *Note:* If *RTData.rta* is located some place other than "*C:\INtimeApps\RTData\Debug*" make the appropriate changes to your code (see the directory name used in the code above).

The last change necessary to complete the application is in the code behind the *Send Data* button. This code consists of a sequence of NTX API calls to retrieve handles for the real-time root process, the *RTData* process, and the data mailboxes, *MY_MBOX_1* and *MY_MBOX_2*. The member variables defined in the *NTXDataDlg.h* header file are being used here to store those handles. Once we have a handle to the data mailboxes, we can send the text typed into the *IDC_DATA Edit Control* to the *MY_MBOX_1 data mailbox* using `ntxSendRtData()`. The last part f the function waits for a return message from *RTData* from the *MY_MBOX_2 data mailbox* using `ntxReceiveRtData()`, and displays the message returned in the *IDC_rxDATA Edit Control*.

6) In the *NTXData* dialog resource, double-click on the *Send Data* button. This will create the empty function into which we can add the code.

7) Insert the following code inside the empty `ICNTXDataDlg::OnBnClickedtxdata()` function and build the solution when you have finished editing the code.

> *Tip:* Open the electronic (PDF) version of this guide and use the Adobe Acrobat "Text Tool" to copy and paste these code fragments directly from the documentation into your Visual Studio project.

```
void CNTXDataDlg::OnBnClickedtxdata()
{
    char        rt_my_mbx_1[] = "MY_MBOX_1";
    char        rt_my_mbx_2[] = "MY_MBOX_2";
    char        rt_TestNTXData_process[] = "RTData";
    NTXHANDLE   hNtx;
    char        send_buf[128];
    char        recv_buf[128];
    int         recv_buf_size;

    //Get a handle to the local INtime node
    hNtx = ntxGetLocationByName("Local");
    //check to see that the INtime kernel is available
    if( ntxGetRtStatus( hNtx ) != E_OK ) {
        MessageBoxEx(NULL,
            _T( "RT Machine not present" ),
```

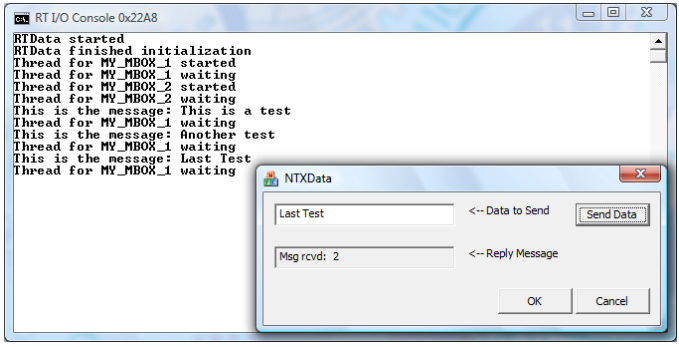
```
            _T( "NTXData"),
            MB_ICONERROR | MB_OKCANCEL, LANG_ENGLISH );
        exit(0);
    }

    //Get root process handle, needed to get RTData process handle
    if( (m_RootProcess = ntxGetRootRtProcess(hNtx))
            == NTX_BAD_NTXHANDLE ) {
        MessageBoxEx( NULL,
            _T( "Could not find INtime root process" ),
            _T( "NTXData" ),
            MB_ICONERROR | MB_OKCANCEL, LANG_ENGLISH );
        exit(0);
    }
    //Get RTData process handle
    if ( (m_TestNTXProcess = ntxLookupNtxhandle(m_RootProcess,
            rt_TestNTXData_process, 0xffff)) == NTX_BAD_NTXHANDLE) {
        MessageBoxEx(NULL,
            _T( "Could not find RTData process" ),
            _T("NTXData"),
            MB_ICONERROR | MB_OKCANCEL, LANG_ENGLISH);
        exit(0);
    }
    //Now get a handle for each mailbox
    if ( ((m_RtMailbox_1 = ntxLookupNtxhandle(m_TestNTXProcess,
            rt_my_mbx_1, 0xffff)) == NTX_BAD_NTXHANDLE)
      || ((m_RtMailbox_2 = ntxLookupNtxhandle(m_TestNTXProcess,
            rt_my_mbx_2, 0xffff)) == NTX_BAD_NTXHANDLE) ) {
        MessageBoxEx(NULL,
            _T( "Could not find data mailboxes" ),
            _T("NTXData"),
            MB_ICONERROR | MB_OKCANCEL, LANG_ENGLISH);
        exit(0);
    }

    //Get the user information typed in IDC_DATA
    //and send it to mailbox MY_MBOX_1
    GetDlgItemText(IDC_DATA, send_buf, 30);
    ntxSendRtData(m_RtMailbox_1 , send_buf, 128);

    //Look for response back from RTData
    if( (recv_buf_size
            = ntxReceiveRtData( m_RtMailbox_2, recv_buf, INFINITE ))
            == NTX_ERROR ) {
        if(ntxGetLastRtError()!=E_TIME) {
            MessageBoxEx( NULL,
                _T( "Received data failed" ),
                _T( "NTXData" ),
                MB_ICONERROR | MB_OKCANCEL, LANG_ENGLISH );
            exit(0);
        }
    }

    //Convert message from ASCI to Unicode
    LPTSTR lpsz = new TCHAR[recv_buf_size +1];
    _tcscpy(lpsz, recv_buf);

    //Update Edit box with value
    SetDlgItemText(IDC_rxDATA, lpsz);
    UpdateData();
}
```

 Copyright © 2005-2009, TenAsys Corporation

Running the Complete Solution

1) Start the INtime kernel.
2) Open INtime Explorer, select the *Local* node, and press *OK*. INtex will show all processes running on the INtime kernel. It will also show any mailboxes associated with those processes.
3) Start *NTXData.exe* within Visual Studio by depressing the *F5* key. Recall that *NTXData* automatically loads and starts *RTData*.
4) After *NTXData* starts you should be able to locate *RTData* in the INtex process tree (remember to enable automatic refresh in the INtex options if you do not see *RTData* appear in the process list).
5) Open the *RTData* process to see the data mailbox objects, *MY_MBOX_1* and *MY_MBOX_2*, and the INtime region object.
6) Type something into the *IDC_DATA Edit Control*, and click the *Send Data* button. Your message will display in the *RTData* console window, and the words Msg r*cvd:* # will appear in the *IDC_rxDATA Edit Control*, where # corresponds to the message sequence number.

Figure 18: Running the Complete Solution



7) Close *NTXData* by pressing either the *OK* or *Cancel* button.

RTData will continue to run, even though you have closed *NTXData*, because we did not include any code to stop *RTData* when *NTXData* terminates.

Use INtime Explorer to shutdown the *RTData* process by right-clicking the *RTData* process icon in the INtime Explorer window and selecting *Delete* from the context menu.

Copyright © 2005-2009, TenAsys Corporation

Example #3: The INscope Performance Analyzer

Determinism is a key attribute of real-time systems. Speed is always a useful attribute to have in any embedded system, but the ability to insure the correct timing and sequence of events can be even more important. This is a key difference between a real-time system and a system that is simply fast.

The INscope performance analyzer is a special software tool that provides you with precise information regarding the timing and sequence of real-time events in a multi-threaded application, so you can measure the determinism of your real-time process. INscope traces events while your application runs in real-time.

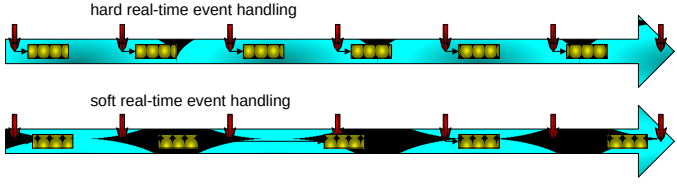
In this section we will use the INscope to monitor a multi-threaded real-time process.

How Fast is Deterministic?

The deterministic nature of a real-time system forces a unique set of requirements upon software applications. A simple definition of a real-time system is one in which the time required to respond to an event is just as important as the logical correctness of that response. Hard real-time systems require the highest degree of determinism and performance. Typically, their worst case event response requirements are measured in microseconds.

Bounded response to events is the key to defining a hard real-time system. Real-time systems require determinism to insure predictable behavior of the system. Without determinism a system cannot be called real-time and without bounded determinism a system cannot be classified as *hard* real-time.

Figure 19: Comparison of Real-time Systems



The specific degree of determinism required is a function of the frequency of the real-time events (size of the time interval between events) and the effect of delays on the dynamic characteristics of that system. That is, how often do events occur and how quick and repeatable must the system be in response to those events. Being able to place a finite and acceptable bound on the value of these numbers

 Copyright © 2005-2009, TenAsys Corporation

is what distinguishes a *hard* real-time system from *soft* real-time systems.

Fast Does Not Equal Deterministic

Faster processors, memory, and peripherals improve the aggregate performance of a system, but they generally do not directly affect the bounded determinism of a system. The worst-case response time to an event may not be significantly changed by using a faster processor; increased speed can decrease the average jitter, the spread and intensity of the variations in response to an event, but it will not eliminate the worst-case jitter.

Improving the performance (or speed) of a real-time system is useful. More performance allows one to increase the complexity of the algorithms that can be implemented in a given period of time (i.e., within a sample interval or cycle). Therefore, the quality of the control and data acquisition system that one can implement in software is improved by using a faster system. However, bounded determinism is still needed to insure that a stable and accurate system, regardless of the performance level, can be deployed.

A Multi-threaded Example

This example application will contain three alarm threads, or fixed interval timing events. Two will be set for the same priority level, and the third will be set one priority level higher.

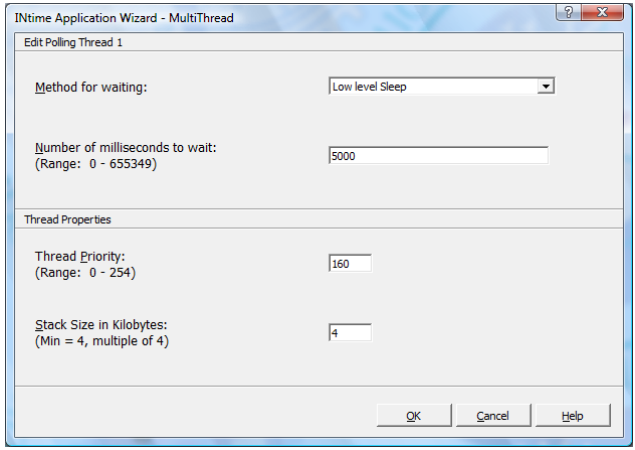
> *Tip:* Complete the *HelloWorld* example before performing this example to familiarize yourself with the INtime development system.

1) Open Visual Studio.
2) Create an INtime project called *MultiThread* and place it in the *INtimeApps* directory you created for the *HelloWorld* example.
3) Select *A full-featured application* from the INtime application wizard dialog and click *OK*.
4) In the next dialog add a *Thread that operates at a regular interval* element.
5) Change the *Method for waiting* parameter from *Sleep* to *Low level Sleep*, change the *Thread Priority* from *170* to *160*, and change the *Number of microseconds* to wait from *1000* to *5000*. This creates *PollThread1* as a thread that will be started on a precise time interval of every five-thousand microseconds (every 5 milliseconds).

Copyright © 2005-2009, TenAsys Corporation

6) Click on *Thread that operates at a regular interval* again. Change the *Method for waiting* parameter from *Sleep* to *Low level Sleep*, change the *Thread Priority* from *170* to *165*, and leave the *Number of microseconds to wait* parameter set at *10000*.

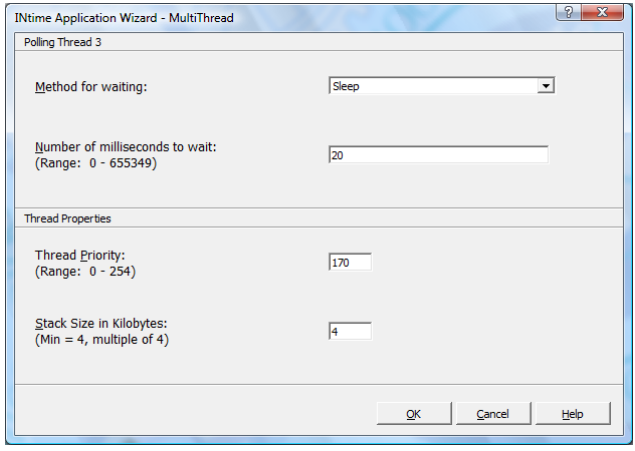
Figure 20: Modifying Thread Parameters



This sets up *PollThread2* as a thread that will be started by the INtime scheduler at a precise time interval of every ten milliseconds.

7) Choose *Thread that operates at a regular interval* a third time. However, this time, specify the following parameters for the thread: *Method for waiting* is *Sleep*, *Number of milliseconds* to wait is *20* and *Thread Priority* is *170*.

Figure 21: Modifying Thread Parameters



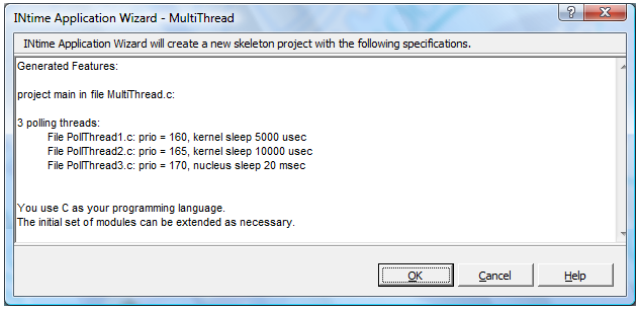
Copyright © 2005-2009, TenAsys Corporation

This sets up *PollThread3* as a simple delay thread, not a precise timer-based interval thread like the previous two threads. As a simple delay thread, *PollThread3* will run approximately once every twenty milliseconds. The imprecision of *PollThread3* is due to the variable amount of processing, especially by higher-priority threads, that can occur between each sleep call.

8) Click *OK*. You now have three time-based threads.

9) Click *Finish* and double-check the summary screen to be sure it lists the following threads and parameters for those threads.

Figure 22: MultiThread Project Summary


INtime Application Wizard - MultiThread
INtime Application Wizard will create a new skeleton project with the following specifications.
Generated Features:
project main in file MultiThread.c:
3 polling threads:
File PollThread1.c: prio = 160, kernel sleep 5000 usec
File PollThread2.c: prio = 165, kernel sleep 10000 usec
File PollThread3.c: prio = 170, nucleus sleep 20 msec
You use C as your programming language.
The initial set of modules can be extended as necessary.
OK
Cancel
Help

10) If everything is fine click *OK* at the summary screen, otherwise push the *Cancel* button, and create a *MultiThread* project that matches the parameters specified above.

11) After clicking *OK*, the wizard will build your project files. Three *PollThread#.c* files will be created. Each *PollThread#.c* file corresponds to one of the three polling thread elements we created using the INtime application wizard.

12) Add the two global variables shown below to the beginning of *MultiThread.c* for communicating between our timing threads.

```
void     PollThread3(void);

DWORD     dwPollThread1;
DWORD     dwPollThread2;

// module variables
```

13) Don't forget to include external declarations in the header file *MultiThread.h* for the two global variables we added above.

```
extern DWORD        dwPollThread1;
extern DWORD        dwPollThread2;
```

Copyright © 2005-2009, TenAsys Corporation

14) To easily identify the three threads in our process from within INtime Explorer and INscope we can add their names to the *MultiThread* process object directory.

    Add the following code to *MultiThread.c*, the file that contains `main()`.

```
CatalogRtHandle(NULL_RTHANDLE,strInit.hPollThread1,"PollThread1");
CatalogRtHandle(NULL_RTHANDLE,strInit.hPollThread2,"PollThread2");
CatalogRtHandle(NULL_RTHANDLE,strInit.hPollThread3,"PollThread3");

// indicate that initialization has finished
strInit.state = INIT_DONE;
```

15) *PollThread1.c* and *PollThread2.c* have nearly identical code. Make the following modifications to each of these files and be sure the variable specified after the TODO line matches the thread number.

```
void            PollThread1(void)
{
#ifdef _DEBUG
    printf("PollThread1 started\n");

#endif
    while (1)
    {
        if (!knWaitForRtAlarmEvent(hAlarmPollThread1,
KN_WAIT_FOREVER))
            Fail("Cannot wait for alarm PollThread1");

//#ifdef _DEBUG
//      printf("PollThread1 waking up\n");
//#endif

        // TODO: do what has to be done every 5000 microseconds
        ++dwPollThread1;
    }
}
```

> *Note:* The code immediately following the `while(1)` statement will differ for each thread, as a function of the time interval and the sleep method specified when you used the wizard to generate the template code. Also, unlike the previous examples, in this example remove (or comment out) the `#ifdef _DEBUG` lines of code inside the `while(1)` statement, we do not want the `printf()` statements to interfere with the output and timing of these threads.

16) *PollThread3.c* contains more code than the prior two. Make the following modifications to this file; again, make sure that the number specified in the `putchar('#')` line matches the thread number.

```
void            PollThread3(void)
{
    int i = 0;
    int x = 0;
```

 Copyright © 2005-2009, TenAsys Corporation

```
#ifdef _DEBUG
    printf("PollThread3 started\n");

#endif
    while (1)
    {
        RtSleep(20);

//#ifdef _DEBUG
//      printf("PollThread3 waking up\n");
//#endif

        // TODO: do what has to be done every 20 milliseconds
        for(i=0; i<10; i++){
            putchar(0x0a);
            for(x=0; x<50; x++)
                putchar('.');
            if(dwPollThread1>0 || dwPollThread2>0) {
                printf(" %.0u %.0u",dwPollThread1,dwPollThread2);
                dwPollThread1 = dwPollThread2 = 0;
            }
        }
    }
}
```

The `for()` loops in *PollThread3* keep the thread alive so we can see pre-emption using INScope. *PollThread1* and *PollThread2* are at higher priorities than *PollThread3*. *PollThread3* can only run when *PollThread1* and *PollThread2* are idle.

17) Make sure the build type is set for *Debug* and compile the project.

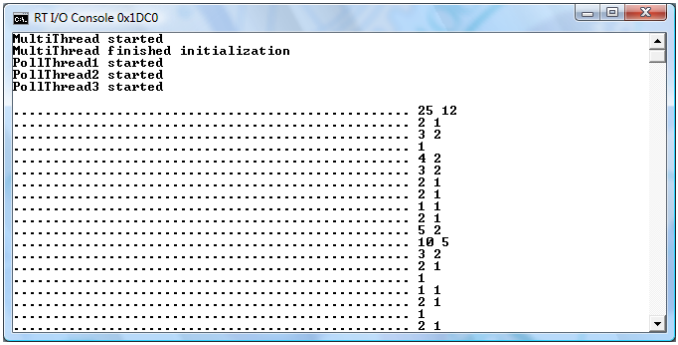
> *Note:* Synchronization code to coordinate setting the values of the two global variables in *PollThread1* and *PollThread2* and reading and resetting those values in *PollThread3* has not been included for ease of instruction.

Trace the Threads With INScope

1) Start INtime Explorer.

2) Using INtime Explorer, start the *MultiThread.rta* application. The application's console window appears, and all three threads start running. *Pollthread3* prints a series of dots in the console window followed by the number of times it was interrupted by the two higher priority threads, *PollThread1* and *PollThread2*.

Figure 23: MultiThread Application Output



3) Start INScope using its shortcut in the INtime programs group (Start | INtime | INScope).

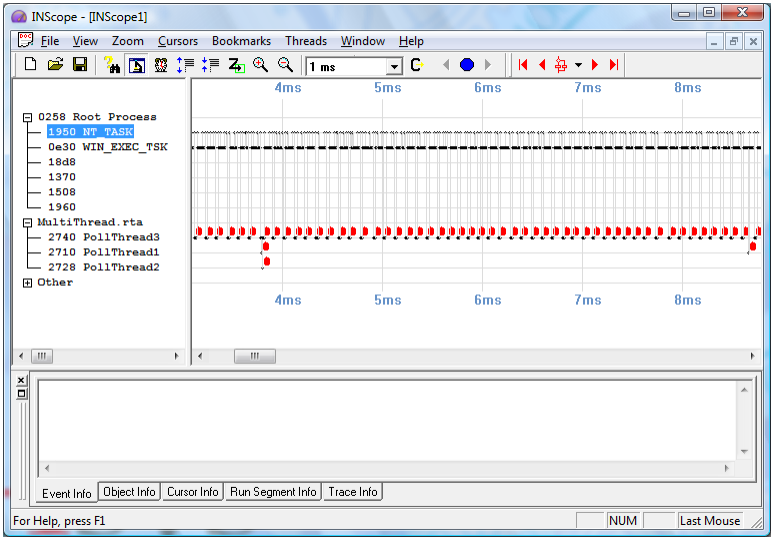
4) The INScope *Trace Control* dialog box appears; click on *Nodes…* and select the *Local* node.

5) While *MultiThread.rta* is running, click *Start Trace*.

Copyright © 2005-2009, TenAsys Corporation

6) In a few moments the *View Trace* button will become visible, indicating that the trace buffer is full. Click the *View Trace* button and the event trace for *MultiThread* is displayed in the INscope upper-right pane.

Figure 24: INscope Event Trace



The left pane lists the INtime processes that were running on the kernel when the trace was started, and each of the threads running inside those processes. The name of the *MultiThread* executable file appears along with the three polling threads, also listed by name. The thread names appear courtesy of the `CatalogRtHandle()` calls we added to `main()`. The exact length of time associated with your trace, and the order of the threads on the display, may differ from the figure above. The time it takes *PollThread3* to run through the `while(1)` loop depends on the speed and configuration of your machine; remember that all INtime `printf()` statements (and other console I/O functions) go through Windows, which will affect some of the timing in this example program.

Scrolling left to right you will see that *PollThread1* and *PollThread2* execute at precise five and ten millisecond intervals, but the timing of *PollThread3* is variable.

Copyright © 2005-2009, TenAsys Corporation

Note: INscope timestamps are derived from your processor's *Time Stamp Counter (TSC)*. If your system has a variable speed clock (such as a laptop with *SpeedStep*) the timing measurements within INscope may be inconsistent. Please visit the TenAsys Knowledge Base at www.tenasys.com for more information regarding this phenomenon.

When all real-time threads are in an idle state the *NT_TASK* and *WIN_EXEC_TSK* threads run. These threads represent Windows, its drivers, applications, and the transfer of information between the INtime kernel and Windows; In shared mode CPU cycles are allocated to Windows only when all of the real-time processes are idle.

The exact set of Windows threads you observe with the INscope tool, and the rate at which those threads run, depends on the number of CPU cores in your system and how your system has been configured. In a single-core system INtime and Windows always share cycles on the single core. In a multi-core system INtime and Windows can share one of the processor cores or INtime can be configured to use one core exclusively for real-time threads. In either case, all remaining CPU cores are allocated to Windows.

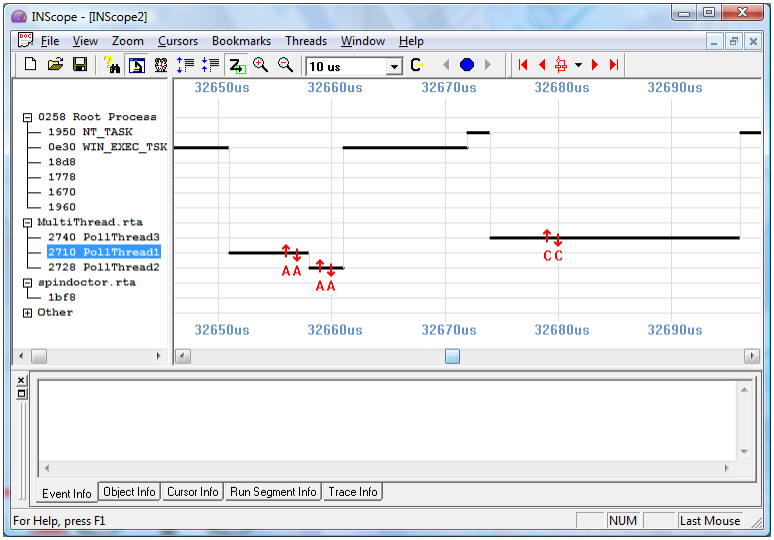
On a single-core processor, had we not included a `sleep()` in the *PollThread3* `while(1)` loop Windows never would have been able to run, and the user interface on your development machine would have frozen; in other words, the system would appeared to have "hung" when, in fact, the real-time threads were simply consuming all the available CPU cycles.

If your development system contains a multi-core processor, Windows would not have frozen if we had omitted the call to `sleep()` in the *PollThread3* `while(1)` loop, because the INtime kernel and its applications will only utilize the resources of one core. Thus, on a multi-core system, Windows always has at least one core available.

 Copyright © 2005-2009, TenAsys Corporation

Use the zoom controls on the menu or toolbar to see the trace more clearly and inspect the task switches between threads. By depressing the 'Z' button on the toolbar and tracing a rectangular region with the mouse you can zoom to a specific segment. In the screenshot below we can see all three threads running in a zoomed view.

Figure 25: Zoomed Inscope Trace

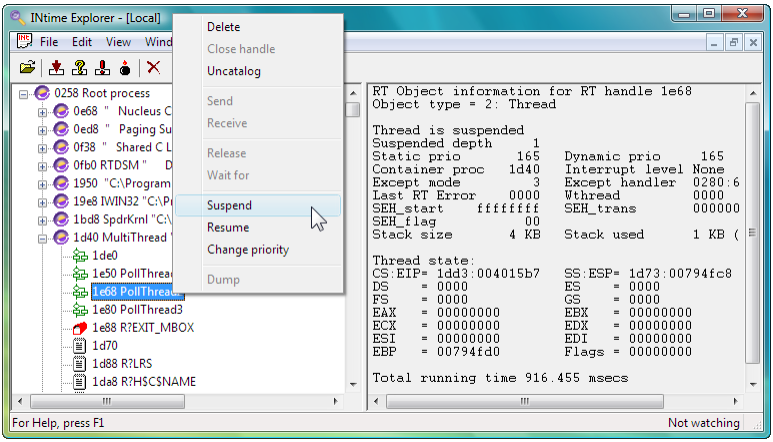


Let your mouse hover over one of the arrows on a 'C' event and you will see it is the `putchar()` function call made inside the *PollThread3* `while(1)` loop. An up arrow is a return from a prior `putchar()` call and the down arrow is a new call into `putchar()`. Since a `putchar()` call results in a transfer of data to the Windows side of the system it forces *PollThread3* into an idle state. Hovering over the 'A' events shows similar information for *PollThread1* and *PollThread2*. Right-click on an event arrow and select the *Display Details* item that appears; data regarding that event is displayed in the *Event Info* tab at the bottom of the screen.

Copyright © 2005-2009, TenAsys Corporation

An interesting and useful feature of INtime Explorer is the ability to suspend and resume threads on the fly.

Figure 26: Intex View of the Multithread App



1) Right-click on the *PollThread2* thread icon, in the INtime Explorer process tree, while *MultiThread* is running (you will have to expand the *MultiThread* process to see its individual threads).
2) Select *Suspend* from the context menu.
3) Note the change in the *MultiThread* console window.

The numbers at the end of each line of dots in the console window indicate how many times each of the two high priority threads ran since the last time the low priority thread ran. These high priority threads can and will pre-empt the low priority thread (as shown by the figure above). If no number appears after the dots it means there were zero precise timer events detected. The numbers vary because the time to run *PollThread3* varies in length.

4) Suspend *PollThread1* and again watch the console window's output.
5) Suspend and resume any of the threads, including *PollThread3*. Do the results match your expectation?

Copyright © 2005-2009, TenAsys Corporation

Next Steps

This guide introduces a variety of INtime development tools and features. The example applications were designed to help you become familiar with developing INtime real-time applications for Windows. The next step is to become familiar with the INtime architecture and API. See the on-line help and *User's Manual* for more detailed information about these subjects.

Once you are familiar with the INtime kernel architecture, you might want to review the sample real-time applications that were installed along with the INtime development package. Appendix C includes a list of the sample applications with their descriptions.

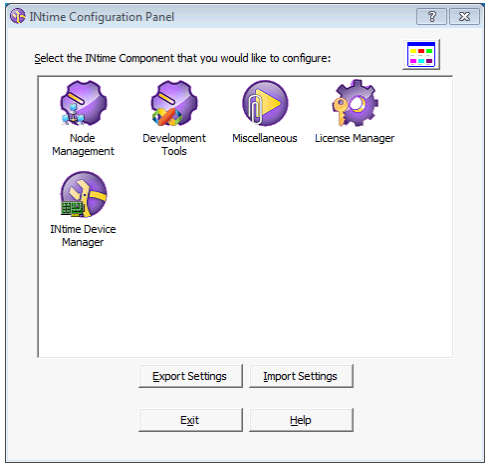
Your final step is to review how to deploy INtime real-time applications. You have the option of creating real-time applications that share the hardware platform with Windows or you can create remote nodes that run the INtime kernel and your real-time application on a standalone system, without Windows. Other areas to explore are deploying INtime on a Windows XP Embedded system (Xpe) and configuring hardware devices, such as network and data acquisition cards, for dedicated real-time access. Please see the documentation for more information.

Copyright © 2005-2009, TenAsys Corporation

A. Configuring the INtime Kernel

The INtime Configuration Applet in the Windows Control Panel can be used to modify run-time parameters in the INtime kernel and your development environment. This appendix describes some of those parameters.

Figure 27: INtime Control Panel Applet



Double-clicking an icon in the applet window starts the individual configuration application.

The *Export Settings* button can be used to save a configuration file from a reference machine that can then be applied to other machines (i.e., for use on a production line) using the *Import Settings* button. The *Export Settings* button will export a single INtime configuration file for all applets that have been selected.

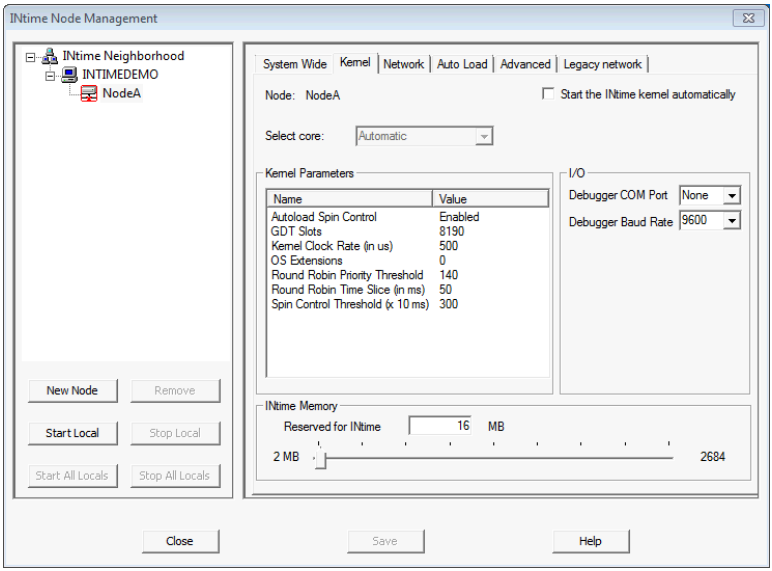
> *Tip:* Hold the *Ctrl* key down while clicking with the left mouse button to highlight multiple configuration applets before exporting your INtime configuration file.

Copyright © 2005-2009, TenAsys Corporation

Node Management

Use this configuration applet to select the best kernel timer rate for your INtime application. In the *MultiThread* example the fastest timer (or alarm) event we could specify was 500 microseconds, because that is the default rate at which the INtime kernel is configured. Changing the *Kernel Clock Rate* to 100 microseconds would have allowed us to create threads that wake with 100 microsecond resolution.

Figure 28: Node Management Kernel Tab



Following are some useful details regarding this applet:

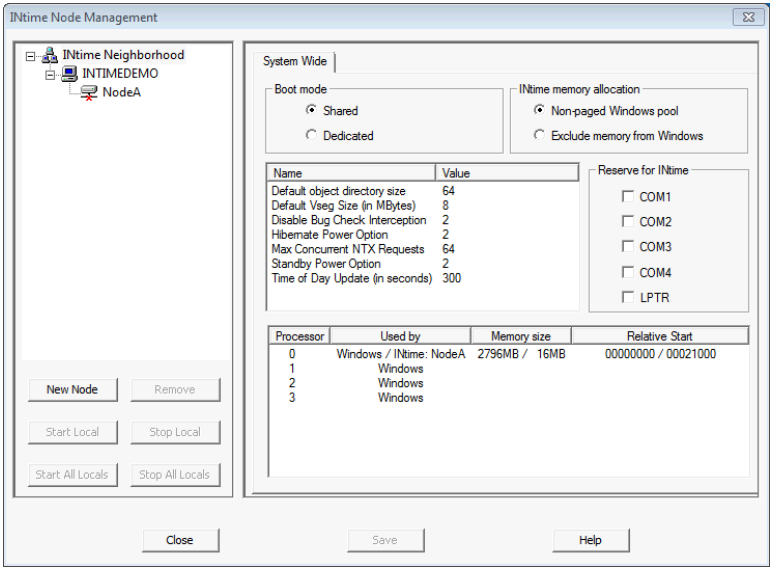
- *Kernel Clock Rate* specifies the number of microseconds that elapse between system clock interrupts. The default is 500 with a range of 100 to 10,000 microseconds.
- *Round Robin Priority Threshold* specifies the priority level at which threads will be scheduled to run using a round-robin schedule. The priority range is 128 to 254. Only threads with identical priorities that are *at or below* the *Round Robin Priority Threshold* are scheduled for round-robin operation.

Copyright © 2005-2009, TenAsys Corporation

Note: INtime priority levels are numbered from 0 to 254, where zero is the highest priority level in the system and 254 is the lowest. Thus, a priority level that is *at or below* the *Round Robin Priority Threshold* means a priority number equal to or higher than that specified as the *Round Robin Priority Threshold*.

- *Round Robin Time Slice* specifies the time allocated for a round-robin time slice. Values range from 20 to 100 milliseconds in multiples of 10 milliseconds.
- *Spin Control Threshold* and *AutoLoad Spin Control* (scroll the *Kernel Parameters* screen down to locate these items) specify the behavior of a special function of the INtime kernel that can be used to detect real-time threads that may be running without pause. In other words, it can be used to identify and stop misbehaving real-time threads that are "locking up" the system.
- *INtime Memory* specifies the total amount of physical memory allocated to the INtime real-time kernel. This is memory that is reserved exclusively for use by the INtime kernel and all real-time processes and threads. This memory is never paged and is, therefore, guaranteed to be deterministic.

Figure 29: Node Management System Wide Tab



Settings which affect all nodes on this host are in the System Wide tab. Following are some useful details regarding this tab:

Copyright © 2005-2009, TenAsys Corporation

- *Boot Mode* specifies how INtime should allocate CPU resources between INtime and Windows: shared or dedicated. Dedicated means one core of a multi-core system is dedicated to an INtime kernel and all real-time applications. Ion this mode you may configure multiple INtime kernels on a multi-core host system. Shared means one core will be shared between INtime and Windows. This configuration parameter has no meaning on a single-core processor, since the CPU is always shared between Windows and INtime when there is only one CPU core.
- *INtime Memory Allocation* specifies where INtime should allocate memory resources from. In the default case INtime allocates memory from the Windows non-paged pool. The amount of memory that can be allocated in this way is limited to 128 Mbytes in order that Windows performance is not restricted. If more memory is required for INtime then the "Exclude memory from Windows" option limits the amount of memory that Windows can access at boot time, and assigns the rest to INtime. This mode may also be used in the case where it is not possible to allocate enough memory because of pool fragmentation.
- *Standby Power Option* and *Hibernate Power Option* specify whether or not INtime should prevent Windows from entering these power management states. Allowing Windows to enter these states will cause the INtime kernel to be shutdown.

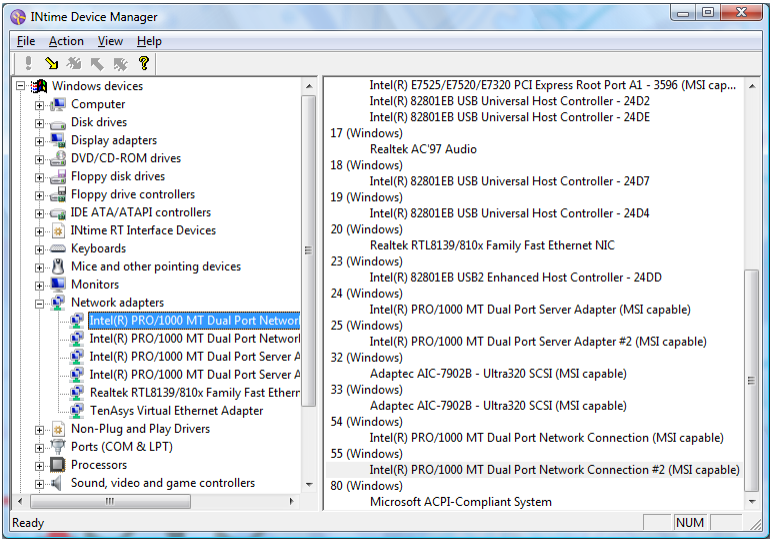
Copyright © 2005-2009, TenAsys Corporation

INtime Device Manager

Use this applet to allocate hardware device resources (especially interrupts) for use by an INtime kernel and your real-time applications. The applet presents a view of all hardware devices in the system, similar to that presented by the Windows Device Manager.

Right-click a device in the list of *Windows devices* and select *Pass to INtime* from the context menu to remove this device from Windows and make it available to the real-time environment.

Figure 30: Device Configuration Applet


INtime Device Manager
File Action View Help
Windows devices
Computer
Disk drives
Display adapters
DVD/CD-ROM drives
Floppy disk drives
Floppy drive controllers
IDE ATA/ATAPI controllers
INtime RT Interface Devices
Keyboards
Mice and other pointing devices
Monitors
Network adapters
Intel(R) PRO/1000 MT Dual Port Networ
Intel(R) PRO/1000 MT Dual Port Networl
Intel(R) PRO/1000 MT Dual Port Server A
Intel(R) PRO/1000 MT Dual Port Server A
Realtek RTL8139/810x Family Fast Etherr
TenAsys Virtual Ethernet Adapter
Non-Plug and Play Drivers
Ports (COM & LPT)
Processors
Sound, video and game controllers
Intel(R) E7525/E7520/E7320 PCI Express Root Port A1 - 3596 (MSI cap...
Intel(R) 82801EB USB Universal Host Controller - 24D2
Intel(R) 82801EB USB Universal Host Controller - 24DE
17 (Windows)
Realtek AC'97 Audio
18 (Windows)
Intel(R) 82801EB USB Universal Host Controller - 24D7
19 (Windows)
Intel(R) 82801EB USB Universal Host Controller - 24D4
20 (Windows)
Realtek RTL8139/810x Family Fast Ethernet NIC
23 (Windows)
Intel(R) 82801EB USB2 Enhanced Host Controller - 24DD
24 (Windows)
Intel(R) PRO/1000 MT Dual Port Server Adapter (MSI capable)
25 (Windows)
Intel(R) PRO/1000 MT Dual Port Server Adapter #2 (MSI capable)
32 (Windows)
Adaptec AIC-7902B - Ultra320 SCSI (MSI capable)
33 (Windows)
Adaptec AIC-7902B - Ultra320 SCSI (MSI capable)
54 (Windows)
Intel(R) PRO/1000 MT Dual Port Network Connection (MSI capable)
55 (Windows)
Intel(R) PRO/1000 MT Dual Port Network Connection #2 (MSI capable)
80 (Windows)
Microsoft ACPI-Compliant System
Ready
NUM

Passing a device to INtime will result in Windows no longer recognizing and loading a device driver for that hardware. Your real-time applications now have exclusive access to the interrupt and hardware registers of that device. This process is needed to insure that Windows drivers and applications will not interfere with your use of the device.

INtime includes support for MSI devices (Message Signaled Interrupts). If you have an MSI capable device you can pass it to INtime even if there is a potential legacy interrupt conflict with Windows by right clicking the device, and selecting the *Pass to INtime (non interrupt or MSI)* menu item.

Copyright © 2005-2009, TenAsys Corporation

B. Debugging With Spider

Debugging the HelloWorld Example

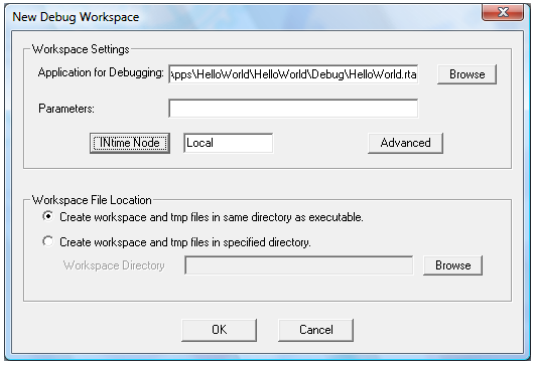
The Spider debugger has a slightly different mode of operation compared with the Visual Studio debugger, which can be useful in some circumstances. When Visual Studio hits a breakpoint, it suspends all of the threads in the process. The Spider debugger can allow them to continue running during the debugging process.

The Spider debugger must also be used when debugging legacy INtime applications generated with older tools, such as Visual C++ 6.0.

The following steps illustrate how to use the Spider debugger with the *HelloWorld* sample application:

1) Start Spider from the INtime program group using Start | All Programs | INtime | Spider Debugger.
2) Select File | New Workspace… Spider creates a workspace file (.idw) to store information for future debug sessions.
3) In the *New Debug Workspace* dialog, find the *HelloWorld.rta* file, using the browse button.
4) Click *Open*.
5) Click on the *INtime Node* button and select the *Local* node.
6) Press *OK*.

Figure 31: Starting a Spider Debug Session

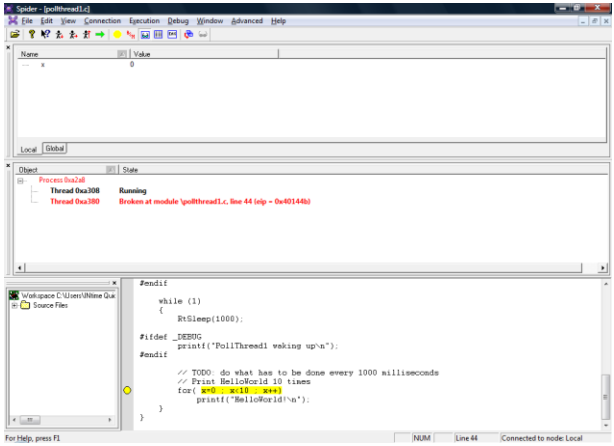


7) The display now changes to show the workspace on the left-hand side of the screen.
8) Expand the *Source Files* icon to display a list of modules.
9) Expand the *PollThread1.c* icon to display a list of public symbols.

Copyright © 2005-2009, TenAsys Corporation

10) Double-click on *PollThread1* to display that function.
11) Set a breakpoint on the `for` loop by clicking somewhere on that line and pressing *F9* (similar to setting break points in Visual Studio).
12) Press *F5* to launch the program.
13) You should see a message like *thread 0x#### has hit a breakpoint*; press *Yes* to switch to the thread that hit the breakpoint.

Figure 32: Reaching a Breakpoint in Spider



You can inspect local and global variables in the upper window. The middle window shows the state of the threads in the process being debugged. The bottom window shows the source of that thread. Press *F10* to single-step, *F11* to step into a call, etc.

Copyright © 2005-2009, TenAsys Corporation

C. Sample Applications

The following table describes the sample applications that are installed with the INtime system. These can be found in the *My Documents\INtime\Projects* folder of the user who installed INtime on your system.

Sample Application	Description
C and C++ Samples for Debugger	The C++ program demonstrates several components of the C++ language available to real-time applications, as well as basic classes, dynamic instantiation, operator overloading, and so on. It also shows the libraries and startup modules needed.
Exception Handling Sample	Includes a Windows and a real-time portion. The Windows portion allows the user to set up timing parameters that control how often a real-time thread will cause a hardware fault. The application demonstrates how another real-time thread can detect and log the failure, delete the offending thread, and recreate it, without affecting Windows or other real-time processes
Floating Point Exception Handling	Demonstrates floating point exception handling.
Graphical Jitter Sample	Measures the minimum, maximum, and average times between low-level ticks using an alarm event handler (precise timer). This application is comprised of both real-time and Windows executables and illustrates use of the NTX API.
Global Objects Sample	Illustrates various aspects of global objects and node management.
High-Performance Ethernet Sample	Illustrates the use of the High-Performance Ethernet drivers included with INtime.
INtime API Sample	Exercises most INtime software system calls.
INtimeDotNet Samples	Sample applications showing the use of the INtimeDotNet assembly for use in Windows apps that use the CLR to communicate to the RT side.
Network Datagrams Sample	Examples of how to send unicast, multicast and broadcast datagrams.
NTX Sample (MsgBoxDemo)	This INtime application has both a Windows and a real-time portion. The Windows portion looks up a mailbox created by the real-time portion and waits at the mailbox. Whenever a real-time thread sends a message to the mailbox the Windows portion displays the received data in a Windows message box. Semaphore and shared memory usage are also demonstrated.

Copyright © 2005-2009, TenAsys Corporation

Sample Application	Description
PCAP Sample Application	Illustrates the use of the PCAP library to filter specific Ethernet packets from the network stack.
Real-time Interrupt Sample	This application illustrates the use of real-time interrupt system calls using the Transmitter Ready interrupt on COM1. A real-time thread takes over COM1 and toggles its Transmitter Ready interrupt. COM1 is disabled when the test ends. Make sure COM1 is available on your system before running this application. When you run the application, continuous activity occurs on the real-time side, preempting Windows for eight 10-second time periods.
RSL Examples	Demonstrates the creation and use of real-time *Shared Libraries*, the INtime analog of Windows DLLs.
Serial Communications Sample	This project demonstrates how to use the INtime Serial Communications library.
TCP Samples	Demonstrates TCP communications between a client and a server. Client and server code is provided for INtime and server code for Windows.
UDP Samples	UDP ping-pong sample application. Datagrams are exchanged between INtime and Windows.
USB Keyboard Sample Client	Demonstrates how to use the INtime USB subsystem by monitoring a USB keyboard and printing a dump of each keystroke as it occurs.
Windows STOP Detection Sample	Shows how an INtime application can detect either a Windows crash (blue screen) or a Windows shutdown event and prevent Windows from completing its normal actions until the real-time application has had a chance to perform a “graceful” shutdown.

Copyright © 2005-2009, TenAsys Corporation